11-2013

Improving model checking stateful timed CSP with non-zenoness through clock-symmetry reduction

Yuanjie SI

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Ting WANG

Improving Model Checking Stateful Timed CSP with non-Zenoness through Clock-Symmetry Reduction

Yuanjie Si[1], Jun Sun[2], Yang Liu[3], and Ting Wang[1]

[1] College of Computer Science, Zhejiang University, China
{siyuanjie,qdw}@zju.edu.cn
[2] Information System Technology and Design,
Singapore University of Technology and Design, Singapore
sunjun@sutd.edu.sg
[3] School of Computer Engineering, Nanyang Technological University, Singapore
yangliu@ntu.edu.sg

Abstract. Real-time system verification must deal with a special notion of 'fairness', i.e., clocks must always be able to progress. A system run which prevents clocks from progressing unboundedly is known as Zeno. Zeno runs are infeasible in reality and thus must be pruned during system verification. Though zone abstraction is an effective technique for model checking real-time systems, it is known that zone graphs (e.g., those generated from Timed Automata models) are too abstract to directly infer time progress and hence non-Zenoness. As a result, model checking with non-Zenoness (i.e., existence of a non-Zeno counterexample) based on zone graphs only is infeasible. In our previous work [23], we show that model checking Stateful Timed CSP with non-Zenoness based on zone graphs only is feasible, due to the difference between Stateful Timed CSP and Timed Automata. Nonetheless, the algorithm proposed in [23] requires to associate each time process construct with a unique clock, which could enlarge the state space (compared to model checking without non-Zenoness) significantly. In this paper, we improve our previous work by combining the checking algorithm with a clock-symmetry reduction method. The proposed algorithm has been realized in the PAT model checker for model checking LTL properties with non-Zenoness. The experimental results show that the improved algorithm significantly outperforms the previous work.

1 Introduction

Timed Automata [2,11] are popular for real-time system modeling and verification. Verification tools for Timed Automata based models have proven to be successful [15,5]. Nonetheless, modeling hierarchical timed systems in Timed Automata is not trivial. The proposed remedies include extensions of Timed Automata [6,8,4,9] or alternative languages [29,16,23]. In our previous work [23], a new language named Stateful Timed CSP (STCSP) is proposed to model hierarchical real-time systems, which combines compositional language constructs from process algebra community (i.e., Timed CSP [20]) with imperative programs and timed constructs like *delay*, *timeout*, and *deadline*, etc. For instance, we write P *timeout*$[d]$ Q in STCSP to denote that process P must perform an action (e.g., a data operation or channel communication) within d

L. Groves and J. Sun (Eds.): ICFEM 2013, LNCS 8144, pp. 182–198, 2013.

time units or otherwise process Q takes over the control and starts executing; we write P *deadline*$[d]$ to denote that P must terminate within d time units.

Like model checking Timed Automata, model checking STCSP models must deal with the emptiness checking problem, i.e., the problem of checking whether a timed model accepts at least one non-Zeno run. An infinite run is non-Zeno if and only if it takes an unbounded amount of time. Zeno runs are infeasible in reality and thus must be pruned during system verification. That is, it is necessary to check whether a run is Zeno so as to avoid presenting Zeno runs as counterexamples. For instance, given a model P *deadline*$[1]$ where $P = a \rightarrow P \mid b \rightarrow Skip$. If property 'eventually event b occurs' is verified without non-Zenoness, then a counterexample with infinitely many a events will be generated. A close look reveals that the counterexample is Zeno since infinitely many a events must occur within 1 time unit. We thus need a method to check whether a run is Zeno or not. Furthermore, the reason that the non-Zenoness checking is particularly interesting is that it is infeasible with zone abstraction [13], which is an effective technique for model checking Timed Automata (which has been employed by many tools including UPPAAL [15]) and STCSP [23]. Zone abstraction, which constructs zone graphs, is too abstract to directly infer time progress and hence non-Zenoness. This issue has attracted much attention recently. The proposed remedies include either introducing an additional clock [24] or additional accepting states in zone graphs [13]. The state-of-art emptiness checking algorithm [12] for Timed Automata has a complexity of $(|C| + 1)^2 \cdot |ZG|$ where $|C|$ is the number of clocks and $|ZG|$ is the size of the zone graph.

Unlike Timed Automata, model checking with non-Zenoness in STCSP can be achieved based on the zone graphs only. It has been shown that zone abstraction can be applied to STCSP, by explicitly associating clocks with the timed constructs, dynamically activating/de-activating clocks and constructing the zone graph [23]. STCSP is different from Time Automata as it relies on implicit clocks which cannot be modified directly. We observe that clocks in STCSP (which are implicit) always have constant upper bounds. For instance, if a clock c is used to model a 'timeout' at time d, c is associated with an upper bound d which will remain constant. Based on this observation, we develop an efficient emptiness checking algorithm for STCSP in [23], based on zone graphs only without introducing extra clocks or states. The algorithm is then applied to model checking STCSP models with the non-Zenoness assumption.

Our Contribution. In this work, we significantly improve [23] by combining the emptiness checking algorithm with a clock-symmetry reduction method. We show that emptiness checking can be performed even without maintaining the clock names (since the clock names in [23] are implicit and 'introduced' anyway), which implies that a clock symmetry reduction method can be applied. That is, the emptiness problem for STCSP can be solved based on a quotient zone graph, which potentially reduces the size of the zone graph by a factor of K factorial, where K is the maximum number of *overlapping* clocks. The experimental results confirm that our method improves previous approaches significantly and allows model-checking with non-Zenoness with minor overheard compared to model checking without non-Zenoness. In summary, we make the following new contributions. First, we develop a clock-symmetry reduction method and combine it with the emptiness checking algorithm for STCSP, which significantly improves the

$$P = Stop \mid Skip \mid e \rightarrow P \mid a\{program\} \rightarrow P \mid \textbf{\textit{if}}\ (b)\ \{P\}\ \textbf{\textit{else}}\ \{Q\} \mid P \mid Q \mid P;\ Q \mid P \setminus X \mid P \parallel Q \mid Wait[d] \mid P\ timeout[d]\ Q \mid P\ interrupt[d]\ Q \mid P\ within[d] \mid P\ deadline[d] \mid Q$$

Fig. 1. Process constructs

performance. Second, we extend the PAT model checker [23] to support LTL checking with non-Zenoness assumption for STCSP, and compare our algorithm with previous approaches by verifying a number of benchmark systems.

2 Stateful Timed CSP

STCSP [23] is a recently proposed real-time modeling language, which extends Timed CSP [19] with shared variables and additional timed process constructs. An STCSP model is a 3-tuple (Var, σ_0, P_0) where *Var* is a set of *finite-domain* global variables; σ_0 is the initial valuation of *Var* (which maps one variable to one value only) and P_0 is a process. A variable can be of a pre-defined type like Boolean, bounded integer, array of bounded integers or any user-defined data type[1]. Process *P* models the control logic of the system using a rich set of process constructs. A process can be defined by the grammar presented in Figure 1. For simplicity, we assume that *P* is not parameterized.

Process *Stop* does nothing but idling. Process *Skip* terminates, possibly after idling for some time. Process $e \rightarrow P$ engages in event *e* first and then behaves as *P*. Note that *e* may serve as a synchronization barrier, if combined with parallel composition. In order to seamlessly integrate data operations, we allow sequential programs to be attached with events. Process $a\{program\} \rightarrow P$ performs data operation *a* (i.e., executing the sequential *program* whilst generating event *a*) and then behaves as *P*. The *program* may be a simple procedure updating data variables (written in the form of $a\{x := 5;\ y := 3\}$) or a complicated sequential program. A conditional choice is written as **if** (b) $\{P\}$ **else** $\{Q\}$. Process $P \mid Q$ offers an (unconditional) choice between *P* and *Q*[2]. Process $P;\ Q$ behaves as *P* until *P* terminates and then behaves as *Q* immediately. $P \setminus X$ hides occurrences of events in *X*. Parallel composition of two processes is written as $P \parallel Q$, where *P* and *Q* may communicate via event synchronization (following CSP rules [14]) or shared variables. Notice that if *P* and *Q* do not communicate through event synchronization, then it is written as $P \mid\mid\mid Q$, which reads as 'P interleaves Q'. Additional process constructs (e.g., while or periodic behaviors) can be defined using the above. In order to focus on the central issue in this paper, we keep the un-timed process compositions minimal and focus only on sequential composition, conditional choice and parallel composition in the following.

In addition, a number of timed process constructs are designed to capture common real-time system behavior patterns. Let $d \in \mathbb{R}^+$. Process $Wait[d]$ idles for exactly *d* time units. Process $P\ timeout[d]\ Q$ behaves as *P* if *P* performs an action before *d* time units

[1] Refer to PAT user manual on how to define a type in C# or Java.

[2] For simplicity, we omit external and internal choices [14] in the discussion.

elapsed since the process starts, or as Q after idling for d time units. Notice that when exactly d time units have elapsed, either P or Q may execute. Process P *interrupt*$[d]$ Q behaves as P for exactly d time units (during the time P may perform multiple actions) and then behaves as Q. Process P *within*$[d]$ constrains that P must *react* (by performing an action) within d time units. Process P *deadline*$[d]$ constrains that P must terminate within d time units. Notice that a timed process construct is always associated with an *integer* constant d which is referred to as its parameter. Furthermore, a process expression Q can be given a name P, written as $P = Q$, and recursion can be defined through process referencing.

Example 1. Let δ and ϵ be two constants. Fischer's mutual exclusion algorithm is a model $(Var, \sigma_0, Protocol)$. *Var* contains one integer variable named *turn* recording the process which attempts to access the critical section most recently. Valuation σ_0 maps *turn* to -1, indicating that no process is attempting initially. Process *Protocol* is defined as $Pro(0) \parallel Pro(1)$ where

$$
\begin{aligned}
Pro(i) = \mathit{if}\ (\mathit{turn} = -1)\ \{ & \\
& (\mathit{set}.i\{\mathit{turn} := i\} \rightarrow \mathit{Wait}[\epsilon])\ \mathit{within}[\delta]; \\
& \mathit{if}\ (\mathit{turn} = i)\ \{ \\
& \quad \mathit{cs}.i \rightarrow \mathit{exit}.i\{\mathit{turn} := -1\} \rightarrow Pro(i) \\
& \} \\
& \mathit{else}\ \{Pro(i)\} \\
\} & \\
\mathit{else}\ \{ & \\
& Pro(i) \\
\} &
\end{aligned}
$$

Process $Pro(i)$ models a process with a unique integer identify i. If *turn* is -1, the process starts attempting to enter the critical section. Firstly *turn* is set to be i (indicating that the i-process is now attempting). Note that this must occur within δ time units (captured by *within*$[\delta]$). Next, the process idles for ϵ time units (captured by *Wait*$[\epsilon]$). It then checks whether *turn* is still i. If so, it enters the critical section. Otherwise, it restarts from the beginning. Mutual exclusion is guaranteed if $\delta < \epsilon$. □

Given a model (Var, σ_0, P_0), its concrete operational semantics is defined through a set of firing rules. We omit the rules here and refer interested readers to [23]. Based on the firing rule, we can systematically construct a labeled transition system (LTS) $CG = (S, init, \Sigma, T)$ such that a state in S is of the form (σ, P) where σ is a valuation function of *Var* and P is a process; $init = (\sigma_0, P_0)$; Σ is the alphabet; and a transition in T of the form $(s, (d, e), s')$ such that $s, s' \in S$ and $(d, e) : \mathbb{R}^+ \times \Sigma$ is the transition label. Note that d is a real number that denotes the time elapsed (since s is reached) before the transition is taken and e is the event name. A (rooted) non-Zeno run of the model is an infinite sequence $\langle s_0, (d_0, e_0), s_1, (d_1, e_1), \cdots \rangle$ of CG such that $s_0 = (\sigma_0, P_0)$ and $(s_i, (d_i, e_i), s_{i+1})$ is a transition of CG for all i and the sum of $d_0, d_1, \cdots$ is unbounded. The sequence $\langle s_0, e_0, s_1, e_1, \cdots \rangle$ is called a timed-abstract run. A model is non-empty if and only if it contains at least one non-Zeno run.

STCSP differs from Timed Automata as it relies on implicit clocks. For instance, intuitively, a clock starts ticking whenever a process *Wait*$[d]$ is activated. Semantically,

it has been shown in [23] that STCSP has the same expressiveness as closed timed safety automata with invisible transitions (i.e., τ transitions), which is strictly less expressive than Timed Automata with invisible transitions and more expressive than closed timed safety automata. More importantly, because clocks in STCSP are implicit and there is no direct way to access the clocks, *the bounds associated with the clocks always remain constant.*

3 Emptiness Checking of Stateful Timed CSP

In the following, we describe the emptiness checking algorithm for STCSP developed in [23]. First we summarize dynamic zone abstraction developed for STCSP and then introduce the emptiness checking approach.

3.1 Dynamic Zone Abstraction

Implicitly, each timed process *instance* is associated with a *unique* clock. For instance, it can be viewed that the clock associated with $Wait[3]$ starts ticking as soon as it is *activated* and expires as soon as it reaches 3. Notice that different instances of the same process are associated with different clocks. For instance, given process $P = Wait[3];\ P$, each invocation of P will generate a different instance of $Wait[3]$ with a different clock. For simplicity, we write $Wait_c[d]$ ($P\ timeout_c[d]\ Q$ and so on) to denote that the associated clock is c. The clock is activated as soon as the process is activated (i.e., the process receives the control). For instance, in process $Wait_{c_1}[5];\ Wait_{c_2}[4]$, only c_1 is activated and c_2 is activated as soon as $Wait_{c_1}[5]$ is terminated. Given a process P, the set of *activated* clocks of P is written as $clock(P)$.

In the abstract zone graph, a node is of the form (σ, P, Z) where σ is a valuation of Var; P is a process; and Z is a zone which is a constraint on values of $clock(P)$. We write $clock(Z)$ to denote the clocks used in Z. For now, we say that two abstraction configurations (σ_0, P_0, Z_0) and (σ_1, P_1, Z_1) are equivalent if $\sigma_0(x) = \sigma_1(x)$ for all x in Var; and P_0 and P_1 are equivalent processes *associated with the same clocks*; and Z_0 and Z_1 share the same canonical form if they are represented as DBMs(Difference Bound Matrices). An abstract transition is of the form $(\sigma, P, Z) \xrightarrow{e} (\sigma', P', Z')$. Notice that $clock(P)$ and $clock(P')$ could be different, i.e., some process constructs may be pruned and some new ones may be activated. For instance, given process $(e \rightarrow Wait_{c_2}[3])\ timeout_{c_1}[4]\ Q$, the transition labeled with e results in a configuration where c_1 is pruned (see the abstract firing rule below) and c_2 is activated. Clocks that are not in $clock(P)$ are irrelevant to future behaviors of (σ, P, Z). Therefore, we always prune those clocks from Z. Given a set of clocks X, we write $Z[X]$ to denote the zone obtained by projecting Z onto X. This operator can be realized based on DBM [23]. Furthermore, we define a function $clean(\sigma, P, Z)$ that returns the abstract configuration (σ, P, Z') where Z' is the conjunction of $Z[clock(P)]$ and $c = 0$ for each $c \in (clock(P) \backslash clock(Z))$. Notice that $Z[clock(P)]$ prunes irrelevant clocks and $(clock(P) \setminus clock(Z))$ contains the newly activated clocks. It can be seen that $clock(P) = clock(Z')$.

The abstract operational semantics is defined through a set of abstract firing rules. We present sample abstract firing rules $P\ timeout_c[d]\ Q$ in the following. The rest can

be found in [23].

$$\frac{(\sigma, P, Z) \xrightarrow{e} (\sigma', P', Z')}{(\sigma, P\ timeout_c[d]\ Q, Z) \xrightarrow{e} (\sigma', P', Z' \wedge c \leq d)}$$

$$\frac{}{(\sigma, P\ timeout_c[d]\ Q, Z) \xrightarrow{\tau} (\sigma, Q, Z^{\uparrow} \wedge c = d \wedge idle(\sigma, P, Z))}$$

The first rule states if a transition of P occurs no later than d time units since the process is enabled, the *timeout* is resolved. Otherwise, if P may delay until $c = d$ (captured by $idle(\sigma, P, Z)$), time out occurs when $c = d$. Function $idle(\sigma, P, Z)$ returns the zone that can be reached by idling from the abstract system configuration (σ, P, Z). For instance, the following shows how the *idle* function is defined for process $P\ timeout[d]\ Q$.

$$idle(\sigma, P\ timeout_c[d]\ Q, Z) = Z^{\uparrow} \wedge c \leq d \wedge idle(\sigma, P, Z)$$

Intuitively, process $P\ timeout_c[d]\ Q$ can idle as long as P can idle and the reading of clock c is less than or equal to d. Similarly, we can define *idle* for all process types. The detailed definition is presented in [23]. *It is important to notice that all the timing constraints are in the form of $c \leq d$ or $c = d$.*

Given a model (Var, σ_0, P_0), using the abstract firing rules, we can build an abstract zone graph $AG = (S, init, \Sigma, T)$ such that S is a set of abstract states (σ, P, Z) such that Z is not empty; $init = clean(\sigma_0, P_0, true)$ is the initial configuration; Σ is the alphabet; and T contains a transition $((\sigma, P, Z), e, clean(\sigma', P', Z'))$ iff $(\sigma, P, Z) \xrightarrow{e} (\sigma', P', Z')$.

It has been shown that *CG* and *AG* share the same set of time-abstract runs [23] and therefore we can model check *AG* against temporal properties like LTL formulae. The number of states in *AG* is bounded by $\#\sigma \times \#P \times \#Z$ where $\#\sigma$ is the number of valuations of *Var*; $\#P$ is the number of process expressions and $\#Z$ is the number of zones. $\#\sigma$ is finite by assumption. $\#P$ is infinite for two reasons. Firstly, due to unbounded recursion, P can be infinitely long. For example, define $P_0 = e \rightarrow (P_0 \parallel P_{new})$ which forks a process P_{new} every time e occurs. The resultant process therefore may contain unboundedly many copies of P_{new}. In this work, we assume that P always has a bounded length, following existing approaches [17]. Secondly, because different timed process instances have different clocks, unboundedly many clocks are used. As a result, there are infinitely many different P and Z. Because P has a bounded length by assumption, there is a bound K on the number of overlapping activated clocks (since every clock is associated with one timed process instance in P). Hence, we can systematically rename the clocks in P (and correspondingly in Z) to a set of reserved K clocks.

Let $C = \{x_1, x_2, \cdots, x_K\}$ be the set of reserved clocks. Given any state (σ, P, Z), for any clock x in $clock(P)$, if $x \notin C$, then x is renamed to an available clock in $C \setminus clock(P)$. As a result, only K clocks are necessary and thus $\#P$ is finite. Lastly, it can be shown that all clocks in STCSP have upper bounds (i.e., $c \leq d$ for all clock c and some integer d) and hence $\#Z$ is finite if the number of clocks is finite. Notice that zone normalization is not necessary. This is exactly the approach proposed in [23]. We refer to the LTS constructed in the above way as *RAG* (short for renamed abstract graph).

3.2 Emptiness Check

In [23], we present an algorithm to solve the emptiness problem based on *RAG*, with a complexity linear in the size of *RAG*. The next theorem reduces the emptiness checking problem to an SCC search problem, the proof of which can be found in [23].

Theorem 1. *A model is non-empty if and only if RAG contains a reachable (maximum) strongly connected component (SCC) scc such that*

† *not all transitions connecting two states in scc are instantaneous; and*
‡ $\{clock(P) \mid (\sigma, P, Z)$ *in scc*$\} = \varnothing$. □

Intuitively, the second condition states that every clock is pruned eventually. The above theorem implies that in order to solve the emptiness problem, we need to test each SCC against two conditions: whether it contains a transition which can be locally delayed; and whether every clock is reset later. Notice that both checks have a complexity linear in the size of the SCC. This leads to the algorithm shown in Algorithm 1. It takes a

Algorithm 1. Previous emptiness checking algorithm for STCSP

```
Algorithm NonEmptinessChecking {
1.      while (there are un-explored states) {
2.          find a new SCC scc;
3.          if (scc satisfies † and ‡) { return true; }
4.      }
5.      return false;
6.  }
```

STCSP model as input, and constructs *RAG* on-the-fly while applying Tarjan's algorithm to identify SCCs. Once an SCC is found, we check whether it satisfies † and ‡. If yes, it returns true at line 3. After checking all SCCs, it returns false. The complexity of the algorithm is linear in time $|RAG|$ (which is due to Tarjan's algorithm for identifying SCC). The overhead of checking † and ‡ is minor.

4 Improved Emptiness Checking Algorithm

In this work, we show that we can improve the performance of emptiness checking using a clock symmetry reduction method.

4.1 Clock Symmetry Reduction

The zone abstraction presented above relies on associating timed processes with explicit clocks. Note that the clock names are irrelevant, except for distinguishing different timed processes. Consider two configurations (σ, P, Z) and (σ, P', Z') such that

$P = Wait_{c_1}[5] \parallel (Q_0\ timeout_{c_2}[6]\ Q_1)$ and $Z = c_2 \leq c_1$
$P' = Wait_{c_2}[5] \parallel (Q_0\ timeout_{c_1}[6]\ Q_1)$ and $Z' = c_1 \leq c_2$

They are exactly the same if we interchange clock c_1 and c_2. In fact, *all clocks are fully symmetric*, since they are implicit and 'introduced' anyway. In the following, we present a method which systematically detects such equivalent states, and potentially reduces the state space by a factor of K factorial. Later, we show that it can be combined with our algorithm for emptiness check.

Observe that there is a fixed ordering on the clocks in $clock(P)$ for any process P, e.g., from left to right as they appear in P. For instance, P as defined above has the sequence $\langle c_1, c_2\rangle$, where P' has the sequence $\langle c_2, c_1\rangle$. Let $\langle c_1, c_2, \cdots\rangle$ be the sequence of clocks in $clock(P)$ with the ordering. Recall that $C = \{x_1, x_2, \cdots, x_K\}$ is the set of reserved clocks. We define a function *map* such that $map(c_i) = x_i$ for all i, i.e., mapping the first clock in the sequence to the first reserved clock x_1 and the second to x_2, etc. In an abuse of notation, we write $map(\sigma, P_1, Z_1)$ to denote the abstract configuration (σ, P_2, Z_2) such that any clock c_i in $clock(P_1)$ is renamed to x_i in P_2 and Z_2. It is easy to see that (σ, P_1, Z_1) and (σ, P_2, Z_2) are equivalent. For instance, the above two configurations are mapped to the same configuration (σ, P'', Z'') such that P'' is $Wait_{x_1}[5] \parallel (Q_0\ timeout_{x_2}[6]\ Q_1)$ and Z'' is $x_2 \leq x_1$.

Given a model (Var, σ_0, P_0) and its abstract zone graph *AG*, we obtain a *QAG* (short for quotient abstract graph) after applying function *map* to every state of *AG*, i.e., an abstract LTS $(S, init, \Sigma, T)$ such that S is a set of abstract states (σ, P, Z) such that Z is not empty; $init = map(clean(\sigma_0, P_0, true))$ is the initial state; Σ is the alphabet; and T contains a transition $((\sigma, P, Z), e, (\sigma', P', Z'))$ if and only if $(\sigma, P, Z) \xrightarrow{e} (\sigma', P'', Z'')$ and $(\sigma', P', Z') = map(clean(\sigma', P'', Z''))$. Lastly, since the clocks in P always appear in the same ordering, we can re-order the clocks in Z such that they follow the same order. *Afterwards, the clock names are irrelevant and the clocks can be anonymized.*

Corollary 1. *AG and QAG are time-abstract bi-similar.* □

Two zone graphs are time-abstract bi-similar if and only if there exists a time-abstract bi-simulation relation between them [23]. In [23], we proved that *CG* and *AG* are time-abstract bi-similar. Corollary 1 can be proved similarly.

4.2 *QAG* Extension

Notice that Theorem 1 requires to check whether a clock is pruned in order to determine whether an SCC is non-Zeno, anonymizing the clocks in *QAG* makes it infeasible to track which clock is pruned. To solve this problem, we extend *QAG* with two transition labels. One is a set of resetting clocks to tell whether a clock is reset later. The other is a Boolean flag to tell whether a transition can be delayed locally. We start with the former.

Let x_0 be another reserved clock, which is not in the set C of reserved clocks presented before. We define a new function *newclean* that satisfies the following: for every (σ, P, Z), $newclean(\sigma, P, Z) = (\sigma, P, Z' \wedge x_0 = 0)$ if $clean(\sigma, P, Z) = (\sigma, P, Z')$. Intuitively, function *newclean* is the same as function *clean* except that it introduces a new clock x_0. The idea is to have a clock at 0 for every state so that by looking at the value of x_0 after a transition, we can infer whether the transition is required to occur immediately. For instance, given $(\sigma_0, Q_0, Z_0) \xrightarrow{e} (\sigma_1, Q_1, Z_1)$ (where x_0 is set to 0 in Z_0), we can infer that the transition must occur immediately if Z_1 implies $x_0 = 0$.

Notice that given a system run in STCSP, a transition that can be *locally* delayed may in fact be constrained to occur immediately *globally*. Consider the following example: $(e \rightarrow \mathit{Wait}[5])$ $\mathit{deadline}[5]$. If we only consider event e, it is only constrained to occur within 5 time units. However, because the process $e \rightarrow \mathit{Wait}[5]$ is constrained to terminate within 5 time units, e must occur immediately.

In the following, we augment *QAG* so as to solve the emptiness problem. Given any transition $((\sigma, P, Z), e, (\sigma', P'', Z''))$ in *QAG*, by definition there exists P' and Z' such that $(\sigma, P, Z) \xrightarrow{e} (\sigma', P', Z')$ and $\mathit{map}(\mathit{newclean}(\sigma', P', Z')) = (\sigma', P'', Z'')$. We associate the transitions with three additional labels. Let $\langle x_1, x_2, \cdots, x_m \rangle$ be the sequence of clocks appearing in P; and $\langle x_1, x_2, \cdots, x_n \rangle$ be the sequence of clocks appearing in P''. Notice that all clocks in P and P'' have been mapped to the set of reserved clocks. The additional labels are:

- a Boolean flag b to indicate whether the transition can be locally delayed.
- a set of indices $R = \{i_1, i_2, \cdots\}$ such that for all i in the set, x_i is in $\mathit{clock}(P)$ but not $\mathit{clock}(P')$, i.e., the indices of clocks which are removed.
- a mapping f such that $f(i) = j$ if $\mathit{map}(x_i) = x_j$ for all $i \notin R$.

A run is then $\langle s_0, (e_0, b_0, R_0, f_0), s_1, (e_1, b_1, R_1, f_1), \cdots \rangle$. With the label R and f, given any state $s_k = (\sigma_k, P_k, Z_k)$ in the sequence, and the i-th clock x_i in the sequence of clocks in P_k, we can check whether x_i is removed later. A clock is removed later if it is removed immediately or renamed to a clock which is removed later. That is, x_i at s_k is removed if and only if there exists $l \geq k$ such that $l = k$ and $i \in R_k$; or $f_k(i) = j$ and clock x_j at s_{i+1} is removed later.

Theorem 2. *Let $\pi = \langle s_0, (e_0, b_0, R_0, f_0), s_1, (e_1, b_1, R_1, f_1), \cdots \rangle$ be a run of QAG. π is non-Zeno if and only if*

- $*$ *there exists infinitely many k such that $b_k = true$;*
- $\star$ *and for all m, every $x_i \in clocks(P_m)$ is removed later.* □

Proof : Recall that every clock is associated with a timed process and every clock is bounded from above. A clock thus puts an upper bound on the execution time of every transition of a segment of the run, i.e., from the moment the clock is activated to the moment the clock is removed.
(***only-if***) If π is non-Zeno, $*$ is trivially true. Since every clock is bounded from above, every clock must be removed since by definition its value goes unbounded along the run; otherwise, we have an empty zone and thus an infeasible run. Hence, $\star$ is true.
(***if***) In the following, we show that if $*$ and $\star$ are true, thus π is progressive [2] and thus non-Zeno. Let the following be a segment of π.

$$\langle s_i, (e_i, b_i, R_i, f_i), s_{i+1}, \cdots, (e_{i+k}, b_{i+k}, R_{i+k}, f_{i+k}), s_{i+k+1} \rangle$$

such that $b_i = \mathit{true}$ and all clocks in the process of s_i are removed before or at the last transition of the segment. Because there are infinitely many such segments, in order to prove that the run is progressive, it is sufficient to show that the segment takes a positive integer amount of time. Let y_j denote the number of time units that can elapse from state s_j to s_{j+1} where $i \leq j \leq i + k$. For each clock c used in the segment (including those

not in s_i), assume that clock c is present at state s_m and not removed until state s_{m+n+1} where $i \leq m \leq m+n \leq i+k$ (i.e., its life-span in the segment). We have a constraint of the following form, which puts an upper bound on the total time of a part of the segment.

$$y_m + y_{m+1} + \cdots + y_{m+n} \sim d_c \qquad -(C1)$$

where $\sim \in \{\leq, =\}$. Because b_i is true, it is implied that $d_c > 0$ if $m = i$ (by assumption $b_i = true$). In the following, we analyze all three cases and show the theorem holds.

- If $d_c = 0$ (which implies $m > i$), $y_m, \cdots, y_{m+n}$ must all be 0. For the constraint on any clock c', we can substitute $y_m, \cdots, y_{m+n}$ with 0 and get a constraint in the same form but with $d_{c'} > 0$. Notice that by $*$, it is guaranteed that not all $d_{c'}$ is 0.
- If $d_c > 0$ for all constraints and if $\sim$ is $=$, then the segment takes at least d (which is a positive integer) time and therefore we conclude that π is non-Zeno.
- If $d_c > 0$ for all constraints and $\sim$ is $\leq$, the constraints are satisfiable with $y_i = d_{min}$ (i.e., y_i equals the minimum of all ds and the rest of the variables equal to 0). Therefore, we conclude that π is non-Zeno.

With the above, we conclude that the theorem holds. □

4.3 Improved Emptiness Check

In the following, we extend Algorithm 1 for STCSP. Notice that emptiness check based on *QAG* is more complicated as we do not maintain clock names (and hence telling whether a clock is removed later is not as straightforward). By Theorem 2, every clock of every state in a run must be checked in order to determine whether the run is non-Zeno or not. The following theorem simplified the task by showing that it is sufficient to check *any* state which is visited infinitely often.

Theorem 3. *A model is non-empty iff QAG contains a reachable (maximum) SCC such that*

† *it contains a transition* $(s,(e,b,R,f),s')$ *where* $b = true$*;*
‡ *and there is a state* (σ,P,Z) *in the SCC satisfies that for every clock in* $clock(P)$*, there is a path from* (σ,P,Z) *in the SCC such that the clock is removed along the path.* □

Proof: (***only-if***) Assume that *QAG* is non-empty, since *QAG* is finite-state, there must be a non-Zeno run and the run must visit a set of states/transitions X infinitely often. There must be an SCC which contains X. X must contain a transition with a label b being true (by contradiction) and therefore † is trivially true. Similarly, every clock of a state in X (which is a state in the SCC) must be removed later (by definition). Thus, there exists some states (σ,P,Z) in the SCC such that every clock in $clock(P)$ is removed later. For every other state (σ',P',Z') in the SCC, because (σ',P',Z') can always reach (σ,P,Z), every clock in $clock(P')$ is removed too (either before reaching (σ,P,Z) or after).
(***if***) Assume there exists an SCC satisfying † and ‡. Let π be a run that visits every

Algorithm 2. Algorithm for STCSP emptiness check

```
Algorithm NonEmpty {
1.  while (there are un-explored states) {
2.    find a new SCC scc;
3.    if (scc satisfies †) {
4.      let s = (σ, P, Z) be a state in scc;
5.      foreach clock x in clock(P) {
6.        if (!IsRemoved(s, x)){
7.          goto line 1; }
8.      }
9.      return true;
10.   }
11. }
12. return false;
}
```

```
Algorithm IsRemoved(s, x) {
1. return false if (s, x) has been explored;
2. foreach transition (s, (e, b, R, f), s') {
3.     if (x ∈ R or IsRemoved(s', f(x))) {
4.         return true; }
5. }
6. return false;
}
```

state/transition in the SCC infinitely often. It is straightforward to see that π satisfies $*$ of Theorem 2 because of †. By ‡, there exists a state s such that every clock c at s is removed later (by visiting s multiple times and each time choosing a path which removes a different clock). For every other state s' in the SCC, there exists a path from s' to s. A clock at s' is either removed before reaching s or removed afterwards (since all clocks at s are removed later). Therefore, $\star$ is satisfied. Thus, π is non-Zeno by Theorem 2 and *QAG* is non-empty.
Therefore, we conclude that the theorem holds. □

The above theorem implies that in order to solve the emptiness problem, we need to check each SCC to see whether it contains a transition that can be locally delayed; and whether all clocks of *any* state are removed later. This leads to the algorithms shown in Algorithm 2. Given a model (Var, σ_0, P_0), algorithm *NonEmpty* constructs *QAG* on-the-fly while applying Tarjan's algorithm to identify SCCs. Once an SCC is found, we check whether it satisfies † (line 3). Lines 4 to 10 then check whether ‡ is satisfied. Notice that at line 4, any state can be picked. For efficiency, we always pick the state in the SCC which has the least number of clocks (since it is sufficient to check any state). The inner loop from line 5 to 8 then checks whether every clock of the state is removed later using algorithm *IsRemoved*. Given a state s and a clock x at the state, algorithm *IsRemoved* returns true if and only if x is removed later. Line 1 of algorithm *IsRemoved* prevents the same pair (a state and a clock) from being explored again, so that the algorithm becomes terminating. Lines 2 to 5 check whether x is removed along any of the outgoing transitions or the renamed clock $f(x)$ is removed at the post-state s' (through a recursive call). If yes, it returns true at line 4. Otherwise, it returns false at line 6. Notice that at line 6 of algorithm *NonEmpty*, if any clock is not removed, then by Theorem 3 there is no non-Zeno run visiting states of the SCC infinitely often and therefore we drop the SCC and go on with checking other unexplored SCCs. If all

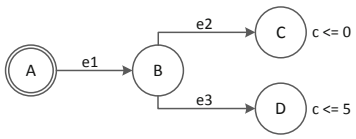

Fig. 2. A general Timed Automaton example

clocks are removed, we return true at line 9 since the run which visits all states and transitions in the SCC infinitely often must be non-Zeno.

The correctness of algorithm can be established based on Theorem 3 straightforwardly. Given that it has been established that *QAG* is finite, it is obvious that the algorithm is terminating. In the worst case, the algorithm runs in time $\mathcal{O}(|QAG|+K^2 \cdot |QAG|)$ where $|QAG|$ is the number of transitions in *QAG* and K is the maximum number of clocks in any state. Firstly, Tarjan's algorithm runs in time $|QAG|$. Secondly, the overhead of checking † is negligible. Lastly, given an SCC in *QAG*, the algorithm for checking ‡ run in time $\mathcal{O}(K_{min} \cdot K_{max} \cdot |SCC|)$ where K_{min} (and K_{max}) is the minimum (and maximum) number of clocks in any state of the SCC and $|SCC|$ is the number of transitions in the SCC. In particular, if there exists a state with no clock (i.e., $K_{min} = 0$), we conclude ‡ is satisfied right away.

In practice, the algorithm performs better than the upper bound complexity for several reasons. Firstly, the algorithm often terminates early as it constructs the state space on-the-fly and terminates as soon as an SCC satisfying † and ‡ is found. Secondly, the overhead of checking ‡ is reduced when (A) we find that a clock is not removed later (in which case we conclude ‡ is not satisfied by the SCC); (B) or we find a transition satisfying a constraint of the form $c = d$ where c is a clock activated at some state in the SCC (in which case we conclude ‡ is true, as at least d time units have elapsed since c is activated). In addition, notice that not all SCCs need to be checked against ‡. For instance, only SCCs which satisfy † (and acceptance conditions from the property, e.g., containing a Büchi accepting state if the property is LTL) are to be checked.

Notice that our approach does not work for Timed Automata in general. Because for every clock in our model, the constraints on the clocks remain the same throughout its life-span, *along any path in the graph*. This allows us to obtain a set of constraints on segments of a run in the form of $(C1)$ (refer the proof of Theorem 2) and also allows us to detect whether a transition can be locally delayed, without referring to a particular path. In the setting of Timed Automata, given any state, the constraints on a clock differ if different paths are taken from the state. For instance, given a Timed Automaton shown in Figure 2, where state *A* is the initial state and *c* is a clock. The transition from *A* to *B* can be delayed given the path from *A* to *D* but not the path from *A* to *C*. Interested readers are referred to [22] on how to extend Algorithm 1 to Timed Automata. This work is however orthogonal to [22] as the clock-symmetry reduction is specific to STCSP.

5 Evaluation

We extend PAT model checker [23] to support model checking with non-Zenoness using our method. We evaluate its efficiency using five examples. The first three are

benchmark systems modeled in STCSP: Fischer's mutual exclusion algorithm, the railway control system, and the CSMA/CD protocol. In addition, we model and verify two hierarchical systems: a simplified pacemaker [3], and a multi-lift system. All models are available online [21]. We remark that modeling the latter two systems in Timed Automata could be non-trivial due to system hierarchy.

The pacemaker example models an electronic implanted device which functions to regulate the heart beat by electrically stimulating the heart to contract and thus to pump blood throughout the body. Quantitative timing is crucial for pacemakers. A pacemaker can operate in many different modes according to the implanted patient's heart problem. We skip the details and refer the readers to [21]. The verified property is that always either a heart beat is detected or the pacemaker eventually stimulates one. The lift system is a standard case study used to demonstrate the power of various specification/verification techniques. It is hierarchical, i.e., the system contains multiple lifts, floors, users, and a central controller; each lift contains a local controller, a button panel; and each local controller is composed of multiple processes (for controlling the shaft, maintaining the request queue, etc.); etc. Furthermore, real-time is an important aspect of the system, e.g., a lift door opens for a certain number of time units; a lift travels at certain speed; etc. The verified property is that a lift door eventually closes.

In our experiments, in order to focus on the reduction obtained using the new method, simple LTL properties which are true are chosen. Notice that because the model checking algorithm is on-the-fly, its performance depends on the searching order in the presence of a counterexample. Table 1 summarizes the experimental results, obtained on a server running 64-bit Windows with Intel Xeon CPU at 2.13GHz and 32GB RAM. Column *RZG* shows the verification statistics based on constructing *RZG* (which renames clocks but not anonymize them). Column K shows the maximum number of overlapping clocks. Column $+Zeno$ shows the verification time *without* the non-Zenoness assumption, and $-Zeno$ shows the verification time *with* the non-Zenoness assumption. Similarly, column *QAG* shows the verification statistics based on constructing *QAG*. Column *OH* shows the overhead of non-Zenoness check and column *Speedup* shows the improvement. Note that '-' means that the data is not available (either out of memory or running for more than 8 hours). The memory consumption in PAT cannot be measured accurately due to limitation of managed memory in .NET framework. The number of states can reflect the real memory usage.

A few observations can be made based on the results. Firstly, it can be shown from the data that non-Zenoness checking incurs some overhead. The theoretical study shows that in the worst case the state space could be enlarged by a factor of $1 + K^2$, whereas in all the experiments, model checking with non-Zenoness takes three times less than the time needed for model checking without non-Zenoness. In average, the overhead is 58% of the verification time without non-Zenoness. Secondly, the experiment results confirm that the verification is significantly faster compared to [23] in all cases, because clock symmetry reduction is combined with non-Zenoness check. The speedup ranges from 2 to 90.25 times. In average, the speedup is 17 for this set of experiments. It can be further noticed that though in theory that the more clocks, the more potential speedup there is, the actual speedup depends on the particular models.

Table 1. Experiment results for STCSP model checking with non-Zenoness

Model	*K*	*RZG*			*QAG*				Speedup	
		States	+Zeno(s)	-Zeno(s)	States	+Zeno(s)	-Zeno(s)	OH	+Zeno	-Zeno
Fischer*5	5	1.1M	180	361	36K	3	4	33%	60	90.25
Fischer*6	6	-	-	-	291K	50	73	46%	-	-
Fischer*7	7	-	-	-	2.6M	1557	4522	190%	-	-
Railway*6	4	158K	14	16	74K	6	6	0%	2.33	2.67
Railway*7	4	1.1M	143	203	527K	44	53	15%	3.25	3.83
Railway*8	4	9.1M	4895	8104	4.3M	818	1339	64%	5.98	13.55
CSMA*6	5	30K	4	5	15K	2	2	0%	2	2.5
CSMA*8	5	237K	46	54	119K	20	21	5%	2.3	2.57
CSMA*10	5	1.6M	1012	1291	803K	239	338	64%	4.23	3.82
Pacemaker	-	-	-	-	1.2M	8711	-	-	-	-
Lift*2*2	4	8.7M	12271	22260	756K	297	728	145%	42.31	30.57

6 Related Work

This work is related to research on hierarchical real-time system modeling and verification. Compositional specification for real-time systems based on timed process algebras has been studied extensively [16,29,18,23]. The closely related work is STCSP [23] which integrates timed process constructs with data variables in order to model complex systems. In [23], zone abstraction, which has been proven successful for Timed Automata, is adopted and used to verify STCSP models. The idea is to explicitly associate clocks with each timed process constructs and use constraints to represent clock values. Furthermore, the approach in [23] is designed to minimize the number of clocks by sharing clocks among all process constructs which are activated at the same time. This work improves [23] by reducing the size of the zone graph significantly (through exploring the symmetry among the clocks). Furthermore, we show that the emptiness problem can be solved based on the reduced zone graph.

This work is also related to research on model checking with non-Zenoness. In [24], it has been shown that zone graphs generated from Timed Automata are too abstract to directly infer time progress and hence non-Zenoness. Syntactic conditions for Timed Automata to be free from Zeno runs have been identified. In [24,27], the authors showed that every Timed Automaton can be transformed into a strongly non-Zeno one, for which, the emptiness problem can be solved easily. The price to pay is an extra clock. Recently, it has been shown that adding one clock may result in an exponentially larger zone graph [13]. The proposed remedy is to transform the zone graph into a *guess zone graph* by introducing extra states. A path of the guess zone graph is non-Zeno if all clocks which are bounded from above are reset infinitely often during the run and the run visits an extra state such that the clocks can be strictly positive [13]. The guess zone graph is $|C| + 1$ times larger than the zone graph and the complexity of the proposed algorithm is $|ZG| \cdot (|C| + 1)^2$ where *ZG* is the size of the zone graph and $|C|$ is the number of clocks. In addition, this work is remotely related to the work on non-Zeno real-time game strategy [7], which however is not based on zone abstraction, whereas

our work is on solving a problem on combining zone abstraction and non-Zenoness. In this work, we show that zone graphs generated from Stateful Timed CSP models are different as all clocks are bounded from above and cannot be reset arbitrarily. As a result, detecting Zeno runs based on zone graphs is feasible. In addition, this work is also related to the work on applying symmetry reduction technique in model checkers, e.g., adding symmetry reduction to UPPAAL [10], exploiting symmetry in RED [28], etc.

In terms of tool support for model checking with non-Zenoness, UPPAAL [15] and KRONOS [5] and RT Spin [26] allow some form of non-Zenoness detection. UPPAAL relies on test automata [1] and leads-to properties. The problem with this approach is that it is sufficient-only. KRONOS supports an expressive language for specifying properties, which allows encoding of a sufficient and necessary condition for non-Zenoness. Checking for non-Zenoness in KRONOS is expensive. The non-Zenoness checking algorithm implemented in RT Spin is unsound [26]. Furthermore, an alternative approach has been proposed by the author in [25]. It is however never implemented. As far as we know, our implementation in PAT is the only model checker that supports model checking LTL with the non-Zenoness assumption.

7 Conclusion

Our contribution in this work is threefold. Firstly, we improve our previous work for STCSP significantly by combining the emptiness checking algorithm with a clock-symmetry reduction method. Secondly, we realize our method in the context of model checking LTL through various benchmark systems and show that it can be used with minor overhead. Lastly, we develop a software toolkit to support model checking LTL with the non-Zenoness assumption.

As for future work, we are investigating how to check timed refinement relationship between two Stateful Timed CSP models with the assumption of non-Zenoness.

Acknowledgment. This work is jointly supported by the project "ZJURP1100105" from Singapore University of Technology and Design, by NSFC Program (No.61103032) and by the National 973 Fundamental Research and Development Program of China under the Grant 2009CB320701.

References

1. Aceto, L., Bouyer, P., Burgueño, A., Larsen, K.G.: The Power of Reachability Testing for Timed Automata. Theoretical Computer Science 300(1-3), 411–475 (2003)
2. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126, 183–235 (1994)
3. Barold, S.S., Stroopbandt, R.X., Sinnaeve, A.F.: Cardiac Pacemakers Step by Step: an Illustrated Guide. Blachwell Publishing (2004)
4. Beyer, D., Rust, H.: Concepts of Cottbus Timed Automata. In: FBT, pp. 27–34 (1999)
5. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A Model-Checking Tool for Real-Time Systems. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)

6. Cattani, S., Kwiatkowska, M.Z.: A Refinement-based Process Algebra for Timed Automata. Formal Asp. Comput. 17(2), 138–159 (2005)
7. Chatterjee, K., Prabhu, V.S.: Synthesis of Memory-efficient "Real-time" Controllers for Safety Objectives. In: HSCC, pp. 221–230. ACM (2011)
8. David, A., Larsen, K.G., Legay, A., Nyman, U., Wąsowski, A.: ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 365–370. Springer, Heidelberg (2010)
9. Dong, J.S., Hao, P., Qin, S., Sun, J.: Wang Yi. Timed Automata Patterns. IEEE Transactions on Software Engineering 34(6), 844–859 (2008)
10. Hendriks, M., Behrmann, G., Larsen, K., Niebert, P., Vaandrager, F.: Adding Symmetry Reduction to Uppaal. Springer (2004)
11. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems. Information and Computation 111(2), 193–244 (1994)
12. Herbreteau, F., Srivathsan, B.: Efficient On-the-Fly Emptiness Check for Timed Büchi Automata. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 218–232. Springer, Heidelberg (2010)
13. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Efficient Emptiness Check for Timed Büchi Automata. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 148–161. Springer, Heidelberg (2010)
14. Hoare, C.A.R.: Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall (1985)
15. Larsen, K.G., Pettersson, P., Wang, Y.: Uppaal in a Nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
16. Nicollin, X., Sifakis, J.: The Algebra of Timed Processes, ATP: Theory and Application. Information and Computation 114(1), 131–178 (1994)
17. Ouaknine, J., Worrell, J.: Timed CSP = Closed Timed Safety Automata. Electr. Notes Theor. Comput. Sci. 68(2) (2002)
18. Reed, G.M., Roscoe, A.W.: A Timed Model for Communicating Sequential Processes. In: Kott, L. (ed.) ICALP 1986. LNCS, vol. 226, pp. 314–323. Springer, Heidelberg (1986)
19. Schneider, S.: Concurrent and Real-time Systems. John Wiley and Sons (2000)
20. Schneider, S., Davies, J., Jackson, D.M., Reed, G.M., Reed, J.N., Roscoe, A.W.: Timed CSP: Theory and Practice. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 640–675. Springer, Heidelberg (1992)
21. Si, Y.J., Sun, J., Liu, Y., Wang, T., Dong, J.S.: Improving Model Checking Stateful Timed CSP with non-Zenoness through Clock-Symmetry Reduction, `http://www.comp.nus.edu.sg/~pat/stcsp`
22. Si, Y.J., Sun, J., Wang, X.Y., Wang, T., Liu, Y., Dong, J.S., Yang, X.H., Li, X.H.: An Analytical Study on Non-Zenoness Checking for Timed Automata. IEEE Transactions on Software Engineering (submitted, 2013)
23. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., André, É.: Modeling and verifying hierarchical real-time systems using stateful timed csp. ACM Transactions on Software Engineering and Methodology (TOSEM) 22(1), 3 (2013)
24. Tripakis, S.: Verifying Progress in Timed Systems. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 299–314. Springer, Heidelberg (1999)
25. Tripakis, S.: Checking Timed Büchi Automata Emptiness on Simulation Graphs. ACM Trans. Comput. Log. 10(3) (2009)
26. Tripakis, S., Courcoubetis, C.: Extending Promela and Spin for Real Time. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 329–348. Springer, Heidelberg (1996)

27. Tripakis, S., Yovine, S., Bouajjani, A.: Checking Timed Büchi Automata Emptiness Efficiently. FMSD 26(3), 267–292 (2005)
28. Wang, F., Schmidt, K.: Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 50–64. Springer, Heidelberg (2002)
29. Wang, Y.: CCS + Time = An Interleaving Model for Real Time Systems. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 217–228. Springer, Heidelberg (1991)